**Вступление**

Всем привет и добро пожаловать в видео, где мы реализуем фуллстек приложение с полноценной авторизацией, при помощи JWT. В нашем клиентском приложении будут содержаться страницы для входа и регистрации нового аккаунта.

При его создании, нам необходимо указать имя и пароль, а также выбрать одну из возможных ролей пользователя - администратор, модератор или обычный пользователь.

После заполнения данных полей и нажатия на кнопку “Зарегистрироваться”, мы мы можем выполнять запросы к защищенным ресурсам. При выполнении такого запроса, сервер проверяет, что мы авторизованы, а также идентифицирует нас и нашу роль на сайте. А при нажатии на кнопку “Выйти” - мы выходим из аккаунта.

Также, мы можем войти и в существующий аккаунт, перейдя для этого на страницу для входа. Здесь мы можем указать имя пользователя и пароль, а затем, нажав на кнопку, снова войти в наш аккаунт.

При создании данного проекта, мы также реализуем возможность авторизации с нескольких устройств и добавим обработку ошибок, в случае ввода некорректных учетных данных. Кроме того, мы реализуем и механизм отслеживания кражи JWT токенов, чтобы ваши приложения получались максимально защищенными!

В этом видео Вы научитесь использовать наиболее необходимые навыки при работе с JWT, которые включают в себя генерирование и правильное хранение JWT как на сервере, так и на клиенте, аутентификация пользователя на сервере по access токену, а также возможность входа в аккаунт с нескольких устройств.

В процессе создания данного приложения мы будем одни из наиболее востребованных на сегодняшний день веб-технологии: PostgreSQL, Express, React и Node, представляющие в совокупности мощный стек PERN.

Возможно сейчас у вас возник вопрос: какие именно начальные знания необходимы перед прохождением этого курса?

Данный курс прекрасно подходит для начинающих. Для его прохождения, Вам лишь необходимо хорошее понимание JavaScript. Знание React, Express и PostgreSQL будет полезным, но не является обязательным. В процессе создания данного фуллстек приложения, мы будем двигаться от простых тем к более сложным.

На данный момент все что Вам необходимо:

1) установленная платформа Node.js актуальной версии,

2) а также любой редактор кода, например VS Code.

Ссылка на финальную версию проекта будет указана в описании к видео. Итак, немного погрузимся в теории, сразу после которой, мы перейдем к разработке!

**Теория**

Итак, прежде чем перейти непосредственно к написанию кода, нам необходимо понять разницу между определениями: аутентификацией и авторизацией, а также разобраться в том, как устроен и работает JWT токен.

Аутентификация - это проверка подлинности пользователя, которая происходит путём сравнения введённого им логина и пароля, с данными, сохраненными в БД.

А авторизация - это проверка прав пользователя на доступ к защищенным ресурсам.

И как же в данных двух процессах участвует JSON Web Token? По сути дела, JWT — это строка, состоящая из трех блоков, разделенных точками: заголовок(header), набор полей (payload) и сигнатура. Первые два блока представлены в JSON-формате и дополнительно закодированы. Набор полей (payload) содержит произвольные пары ключ/значения, а подпись, может генерироваться при помощи алгоритмов шифрования

Например, так, может выглядеть подписанный JWT токен (после декодирования 1 и 2 блоков):

{ alg: "HS256", typ: "JWT" }.{ iss: "auth.myservice.com", aud: "myservice.com", exp: 1435937883, userName: "John Smith", userRole: "Admin" }.S9Zs/8/uEGGTVVtLggFTizCsMtwOJnRhjaQ2BMUQhcY

Итак, после успешного входа в аккаунт, мы получаем пару токенов - accessToken и refreshToken.

access token - используется для авторизации запросов и хранения дополнительной информации о пользователе (например id пользователя и его роль, такую информацию также называют payload). Все поля в payload могут хранить абсолютно любые данные. То есть, в текущем случае, поля user\_id и role не являются обязательными. Здесь это лишь частный случай.

refresh token - выдается сервером в результате успешной аутентификации и используется для получения новой пары access и refresh токенов. Более подробно про его работу, мы поговорим чуть позже.

Итак, каждый токен имеет свой срок жизни, например access токен - в среднем от 15 мин до 1 часа, а refresh токен - от 7 до 60 дней.

И поскольку токены - это не зашифрованная информация - крайне не рекомендуется хранить в них какую либо sensitive data (например пароли, данные для оплаты и т.д.)

Получив данные два токена от сервера, мы сохраняем их в клиентском приложении. refreshToken мы поместим в httpOnly cookie, для того, чтобы исключить возможность получения ее из js кода, а accessToken - сохраним не в localStorage как это обычно делают, а в памяти клиентского приложения. Чуть позже, мы разберем, для чего это необходимо.

В будущем мы будем выполнять запросы к конечным точкам, доступ к которым будет только у авторизованных пользователей.

И тут возникает вопрос. А как веб-серверу понять, что у нас действительно есть доступ к такой конечной точке? Для того, чтобы сервер смог нас идентифицировать, в такой http запрос нам необходимо приложить accessToken. Мы добавим заголовок Authorization, в котором передадим accessToken на сервер. При получении данного запроса, сервер авторизации проверит валидность accessToken, и в случае успешной проверки, перейдет к обработке запроса.

Если токен не прошел проверку, то сервер отклонит запрос от клиента.

На этом с теорией все, давайте перейдем к реализации.

**Скачивание бэкенда и фронтенда**

Перед началом работы, пожалуйста склонируйте указанный в описании репозиторий. В нем содержатся начальные версии как фротенд, так и бэкенд приложения. Нам лишь необходимо реализовать в них логику авторизации и аутентификации для клиентской и серверной сторон.

Для того, чтобы склонировать данный репозиторий, мы можем выбрать опцию Clone Git Repos в VS Code. И в появившемся поле - указать данную ссылку. Затем - указать место на компьютере, куда его необходимо склонировать, например, папку auth, и дождаться окончания скачивания.

**Установка пакетов**

Итак в нашей корневой директории auth содержатся две папки - client и api - для работы с фронтендом и бэкендом соответственно. Прежде чем реализовать код для авторизации и аутентификации, нам необходимо установить необходимые зависимости. Для этого, давайте откроем терминал во вкладке View, а далее нажмем на кнопку Split в правом верхнем углу, чтобы разделить наш терминал на две части. Далее, устанавливаем пакеты в обе директории при помощи команды npm i.

В клиентском приложении, мы будем использовать зависимость axios для выполнения http запросов.

А в серверном приложении, нам понадобятся зависимости bcryptjs для хэширования паролей пользователей, cors, dotenv для работы с переменными окружения, express, jsonwebtoken, nodemon для автоматического перезапуска сервера, pg для работы с postgres, yup для валидации данных в запросах, а также fingerprint, о котором мы поговорим чуть позже.

**Запуск приложений**

После установки пакетов, давайте запустим оба проекта при помощи команды npm run dev.

Сейчас, если мы перейдем по данному адресу в браузере, то мы увидим фронтенд приложение, которое состоит из трех страниц. На первой странице мы можем войти в аккаунт, введя для этого имя пользователя и пароль, а затем нажав Enter.

Имя является уникальным значением, которое пользователь придумывает самостоятельно. Вместо данного поля, здесь может быть и адрес электронной почты, ник пользователя и т.д.

Также, на данной форме, имеется базовая валидация, на случай пустого ввода или ненадежного пароля. В учебных целях, минимальную длину пароля мы укажем равной 3 символам. Конечно, в неучебных проектах, проверка сложности пароля должна быть реализована более тщательным образом.

Пока что, при нажатии на кнопку, вход, ничего не происходит.

Следующая страница - это создание аккаунта. Она имеет такую же форму, как и на странице для входа, но с единственным дополнением - возможностью выбора роли пользователя. В данном проекте, мы реализуем 3 возможные роли - администратор, модератор и обычный пользователь.

После входа в аккаунт, мы будем попадать на страницу авторизованного пользователя. При нажатии на данную кнопку, мы будем выполнять запрос на защищенный эндпоинт сервера, и затем выводить полученную информацию здесь, на экране.

И наконец, на данной странице, мы также можем и выйти из аккаунта.

С точки зрения реализации, в нашем клиентском приложении имеются 3 компонента, для каждой из представленных страниц, а также контекст, для управления авторизацией на клиенте - AuthContext. В нем содержатся 4 функции - handleFetchProtected - мы будем вызывать ее для выполнения запроса на защищенный ресурс, handleLogOut - она будет вызвана при нажатии на кнопку “Выйти”, а также handleSignUp - при нажатии на кнопку зарегистрироваться.

Здесь, в data, мы будем получать объект введенными значениями - userName и password. И последняя функция - это handleSignIn - она будет вызвана при нажатии на кнопку “Войти”. И здесь мы также получаем поля формы внутри объекта data.

Сейчас, давайте перейдем в папку с серверным приложением. Здесь, в главном файле server.js, мы запускаем http сервер на порте 5000. Чуть выше, мы добавляем express.json, чтобы наш сервер мог обрабатывать формат JSON в запросах, чуть ниже, мы добавляем миддлвейр cors, а также cookieParser, для работы с файлами куки.

Кроме того, при получении каждого запроса от клиента, мы будем генерировать fingerPrint. Что же это такое? Фингерпринт (он же отпечаток браузера) — это уникальный набор данных об устройстве, с которого пользователь просматривает веб-страницу в Интернете. О том, как и зачем его использовать, мы поговорим чуть позже.

Внутри cors, давайте сразу добавим конфигурации. Мы добавим свойство credentials и запишем в него значение true, а также в свойстве origin мы укажем адрес нашего фронтенд приложения - <http://localhost:5173>. Давайте сразу вынесем его в переменную окружения, мы сохраним ее как CLIENT\_URL и, затем, укажем в файле server.js.

Итак, здесь, для пути /auth, мы указываем AuthRootRouter. В данном роутере находятся конечные точки, необходимые для работы авторизации. Их всего 4 - логин, регистрация, выход из аккаунта и рефреш - необходимый для получения новой пары аксес и рефреш токенов.

Обработку запросов для каждой из конечных точек, мы реализуем чуть позже, в соответствующих контроллерах класса AuthController. Также, для каждого запроса имеется валидация, реализованная при помощи зависимости yup, а в файле Errors папки utils, описаны основные классы ошибок, а также метод для их обработки.

**Регистрация в ElephantSQL**

И перед тем, как перейти к реализации логики бэкенда, нам необходимо подключить базу данных к нашему проекту. В данном видео мы воспользуемся PostgreSQL, для работы с которым, мы будем использовать веб-сервис ElephantSQL, который предоставляет облачное хранение для наших данных. Ссылка на данный ресурс находится в описании к этому видео.

Переходим в него и, авторизовавшись, например, с помощью гугл аккаунта, кликаем на кнопку Create New Instance. Далее, указываем название базы данных. Например, auth. Здесь можно выбрать наиболее близкий регион, и, далее, убедившись в правильности введенной информацию, нажимаем Create Instance.

Прекрасно! Теперь мы можем подключиться к данному экземпляру внутри нашего серверного приложения. Для этого копируем сгенерированное название пользователя и помещаем его в свойства user и database конфига, расположенного в файле db.js. Также переносим значение полей password и host. Прекрасно!

**Регистрация (фронтенд)**

Итак, мы начнем с регистрации пользователя. Здесь, в компоненте SignUp, где находится форма регистрации, вызывается функция handleSignUp при отправке данной формы. Данная функция находится в контексте AuthContext и получает объект data - введенные пользователем данные. Данный объект содержит три поля - userName, password и role, являющуюся числовым значением, в диапазоне от 1 до 3 включительно. И прежде чем отправить их на сервер, мы создадим экземпляр axios, чтобы работать с авторизацией.

Мы укажем здесь const AuthClient = axios.create() и внутрь передадим объект, где мы укажем настройки для данного экземпляра. Мы добавим объект со свойством baseURL, в которое мы поместим значение <http://localhost:5000/auth>, а также укажем свойство withCredentials, равное true, чтобы разрешить отправку куки. Теперь, нам необходимо отправить запрос на регистрацию.

Мы укажем здесь Auth.post и внутри круглых скобок передадим слеш sign-up, а через запятую, укажем тело запроса - объект data.

Теперь, если мы откроем панель разработчика и перейдем на вкладку сеть, а затем введем данные в форму и нажмем на кнопку “Зарегистрироваться”, то мы увидим отправленный нами запрос на сервер. Мы видим его url, а также тело запроса, содержащее введенные значения для полей userName, password и role.

**Регистрация (бэкенд)**

Прекрасно! Сейчас, давайте перейдем в серверное приложение, в файл Auth.js, расположенный в папке routers. Здесь, в роутере, мы видим данный пост запрос sign-up. Он проходит валидацию, проверку на то, что приходящие поля заполнены и, в случае, если данные, приходящие от клиента, корректны, будет вызван метод signIn, расположенный в классе AuthController.

В данном проекте мы будем использовать архитектурный паттерн "Controller-Service-Repository".

Основная идея данного паттерна заключается в том, что наше приложение разделено на три составляющих: контроллеры (Controllers), сервисы (Services) и репозитории (Repositories).

Когда на сервер приходит HTTP запрос, происходит вызов соответствующего контроллера. Он обрабатывает запрос, используя сервисы для выполнения бизнес-логики. Сервисы, в свою очередь могут использовать репозитории для доступа к данным. С помощью репозиториев, происходит выполнение операций чтения и записи данных в хранилище. В нашем случае — это работа с postgres.

Теперь здесь, в блоке try, мы вызовем метод signUp класса AuthService. Давайте деструктурируем данные из тела запроса, здесь сверху. Мы укажем const { userName, password, role } = req.body;

Так же мы получим значение fingerprint из запроса: const { fingerPrint } = req;

Далее, внутри блока try, мы добавим вызов await AuthService.signUp, передав в него внутри объекта значения userName, password, role и fingerprint..

В данном методе signUp, мы реализуем основную логику для регистрации пользователя. Формально, данную логику можно представить следующим образом:

1. Пользователь регистрируется в приложении, передавая на сервер имя, пароль и роль.

2. Генерирует на сервере fingerprint браузера (или некий иной уникальный идентификатор устройства, в случае, если это не браузер)

3. Проверяем, существует ли в пользователь с данным именем в базе данных.

4. Если такой пользователь существует - выбрасываем ошибку.

5. Хэшируем введенный пароль

6. Добавляем запись в базу данных о текущем пользователе (поля: имя пользователя, хэшированный пароль, роль), получаем id пользователя

7. Генерируем access и refresh токены

8. Добавляем рефреш-сессию в базу данных (поля: id пользователя, refresh токен, fingerprint браузера) (об этом немного позже)

9. Записываем refresh токен в httpOnly cookie

10. Возвращаем клиенту access токен

Здесь может возникнуть вопрос по поводу 8 пункта - что такое рефреш-сессия и для чего она используется.

Для использования возможности аутентификации на более чем одном устройстве, нам необходимо хранить все рефреш токены по каждому пользователю.

Данный список мы сохраним в базе данных. В процессе каждого логина или регистрации пользователя, мы будем создавать запись с Fingerprint его браузера и рефреш-токена. Такую информацию мы назовем рефреш-сессией.

Рефреш токен на сервере, мы также храним для учета доступа и инвалидации краденых токенов. Таким образом сервер точно знает о клиентах которым стоит доверять (кому позволено авторизоваться). Если не хранить рефреш токен в БД то велика вероятность того что эти токены будут бесконтрольно гулять по рукам злоумышленников, для отслеживания которых нам придется создавать черный список и периодически чистить его от просроченных токенов. Вместо этого мы храним лимитированный список белых токенов для каждого пользователя отдельно и в случае кражи, у нас будет механизм противодействия, который мы реализуем позже.

Итак, в методе signUp класса AuthService, первым делом, нам необходимо проверить, что пользователя с данным ником еще нет в базе данных. Мы укажем здесь const userData = await UserRepository.getUserData и внутрь передадим значение userName. Пока что, данный метод, еще не реализован, но скоро, мы это исправим.

Итак, если пользователь с данным именем уже существует, то есть if userData, то мы выбросим исключение и вернем статус 409 - конфликт. Мы укажем throw new Conflict() и внутрь передадим текст “Пользователь с таким именем уже существует”.

Вместо того, чтобы хранить пароль в обычном виде, нам необходимо его захешировать. Без хеширования, пароли, хранящиеся в базе приложения, могут быть украдены, например, если база данных была скомпрометирована.

Применяя хеширующий алгоритм к паролям, перед сохранением их в базе данных, мы делаем невозможным разгадывание оригинального пароля. Иными словами, мы не можем получить исходный пароль из хэшированного. Поэтому, при входе в аккаунт, мы будем сравнивать хэши пароля, введенного пользователем с хэшем, хранящимся в базе данных.

Для того, чтобы захешировать пароль, мы воспользуемся зависимостью bycript.

Мы создадим новую константу const hashedPassword = bycrypt.hashSync(password) "," и укажем количество раундов, к примеру, равным 8.

И прежде чем перейти к дальнейшей логике, давайте реализуем метод getUserData в классе UserRepository. Переходим в файл User.js, расположенный в папке repositories.

Для хранения информации о пользователях, в нашей базе данных будет содержаться таблица Users. И здесь, в файле queries.sql, мы можем наблюдать структуру такой таблицы. Она состоит из трех полей - id пользователя, который будет генерироваться автоматически, при добавлении нового пользователя, поле name, которое будет уникальным для каждого пользователя и поле password - хешированный пароль.

Итак, теперь давайте перейдем в ElelphantSQL, во вкладку Browser, где мы укажем данный запрос. Нажимаем Enter и через некоторое время, мы видим, что данный запрос был успешно выполнен, таким образом, данная таблица была создана.

Переходим в файл User.js. И здесь, в методе getUserData, мы добавим запрос в базу данных. Мы укажем const response = await pool.query(“SELECT \* FROM Users WHERE name = $1”, и далее, вторым аргументом, передадим массив, в котором мы укажем всего одно значение - userName.

Ответ на запрос в базу данных содержится в свойстве rows. Данное свойство является массивом, содержащим все записи, которые удовлетворяют выполненному запросу. Если длина такого массива равна нулю, то пользователь с данным с данным именем отсутствует. Мы добавим условие if (!response.rows.length), и в этом случае, мы вернем значение null. В противном случае, мы вернем информацию о найденном пользователе, которая содержится в массиве rows, и, так как поле имя уникальное, то мы вернем единственный элемент, под индексом 0.

Следующим шагом, мы добавим нового пользователя в базу данных. Мы укажем здесь await UserReposirory.createUser() и, в объекте, передадим значения userName, hashedPassword и role.

Внутри данного метода, мы укажем SQL запрос, чтобы добавить запись о пользователе. Мы скажем await pool.query(“INSERT INTO Users (name, password, role) VALUES ($1, $2, $3) RETURNING \*”), далее, вторым аргументом мы передадим массив, в котором по порядку укажем значения userName, hashedPassword и role.

Теперь, нам необходимо получить созданного пользователя и вернуть его в результате работы метода. Для этого, мы укажем return response.rows[0].

Данный id мы деструктурируем в сервисе. И сейчас, чуть ниже, давайте создадим константу payload, в которую мы поместим значения id, userName и role. И теперь, мы сгенерируем access и refresh токены с данным набором полей.

Для этого, в классе TokenService, давайте реализуем два метода - generateAccessToken.js и generateRefreshToken.

Перед их реализацией, нам необходимо создать секретные ключи и сохранить их на сервере.

С их помощью мы будем формировать подписи для токенов. Таким образом наши итоговые токены будут представлять собой закодированную и подписанную строку. И далее, при каждом новом запросе к защищенным конечным точкам нашего приложения, мы сможем проверить такую подпись при помощи секретного ключа, чтобы убедиться в том, что токен не был изменен.

Для генерации такого ключа, можно, например, воспользоваться зависимостью crypto, создать случайную последовательность байт, а затем перевести ее в формат base64.

Мы же просто создадим произвольные 2 строки в файле .env. Здесь главное то, что данные две строки хранились строго на сервере. Мы добавим здесь ACCESS\_TOKEN\_SECRT и refresh ts соответственно. Значения, находящиеся в них не так важны, как тот факт, что данные значения должны храниться строго на сервере

Модуль dotenv используем для загрузки переменных окружения из файла .env. (.config)

Внутри метода genAT, мы вызовем метод sign, который создаст JWT на основе переданного объекта payload. Вторым аргументом нам необходимо передать значение ACCESS\_TOKEN\_SECRET, чтобы создать подпись для токена. И наконец, третьим аргументом, мы передадим объект с настройками токена. Мы укажем свойство expires, равное 30 минутам - время жизни access токена.

Добавляем return, и ключевое слово await, так как данная операция асинхронная. И, таким образом, данный метод будет создавать и возвращать accessToken, подписанный с использованием секретного ключа ACCESS\_TOKEN\_SECRET.

Давайте скопируем данный код и перенесем его в метод generateRefreshToken. Здесь мы оставим все то же самое, только в качестве секретного ключа мы передадим значение REFRESH\_TOKEN\_SECRET, а срок действия такого токена, мы укажем равным 15 дням.

И наконец, мы создадим рефреш-сессию для пользователя, регистрирующегося с данного устройства (с данным finger\_print). Мы укажем await RefreshSessionsRepository.createRefreshSession(), и внутрь, в объекте передадим 3 свойства - id пользователя, refreshToken и fingerPrint пользователя. В данном методе репозитория, мы деструктурируем эти поля и указываем запрос await pool.query(“INSERT INTO RefreshSessions (user\_id, refresh\_token, finger\_print) VALUES ($1, $2, $3)”) и затем, в массиве, мы передадим данные 3 значения по порядку - id, rt, fp.

В результате работы данного сервиса, мы вернем созданные accessToken, refreshToken и значение accessTokenExpiration, упаковав их в объект. В значение accessTokenExpiration, мы поместим одноименную константу, расположенную в файле constants. Каким образом формируется данное значение? Мы указываем его в миллисекундах, и оно равно 30 минутам, умноженным на 60 секунд, умноженным на одну 1000.

Теперь, давайте вернемся в контроллер и здесь, мы получим информацию о токенах от сервиса, мы деструктурируем ее значения. И в итоге, значения accessToken и accessTokenExp мы отдаем клиенту, в формате JSON, а значение refreshToken записываем в куки.

Мы укажем res.cookie(“rt”, rt,) и далее, мы передадим конфигурации куки - значение COOKIE\_SETTINGS.REFRESH\_TOKEN, также из файла constants. Давайте посмотрим, как они выглядят. Мы используем свойство httpOnly: true, чтобы даннаякуки не была доступна из js-кода, maxAge равен сроку жизни куки в миллисекундах (в данном случае - это 15 дней, умноженные на 24 часа, умноженные на 3600 секунд, умн. на 1000. В значение path мы указываем пути /auth/logout и /auth/refresh, чтобы данная куки отправлялась на сервер только при выходе из аккаунта или при получении нового аксес токена. И наконец, в sameSite мы записываем значение strict, чтобы данная куки отправлялась только на данный сервер. В дальнейшем, уже при наличии https, необходимо также добавить и флаг secure, равный true.

Если сейчас мы вновь вернемся во фронтенд приложение и попытаемся зарегистрироваться, введя для этого имя пользователя и пароль, а затем нажмем на кнопку ниже, то в панели разработчика мы увидим данный запрос, который возвращает нам объект с 2 свойствами: accessToken и ate-n, также, в заголовках, мы видим refreshToken в куки, переданный с указанными на бэкенде полями. Браузер сохранил данную куки, и теперь данный rt, также отображается во вкладке приложение, в файлах куки.

Если мы зайдем в elephantSQL, во вкладку браузер, и выполним запрос для получения всех записей о пользователях, при помощи SELECT \* FROM Users; и далее нажмем execute, то мы увидим информацию о новом пользователе - его id, имя и захешированный пароль.

А при получении всех записей в таблице RefreshSessions, мы видим id данного пользователя, а также его finger\_print и refresh\_token.

Все, что нам остается сейчас сделать - это правильно сохранить accessToken на стороне клиента. RefreshToken уже автоматически поместился в куки.

**Управление access токеном на клиенте**

Нам нужно сохранить наш access токен так, чтобы мы могли отправлять его в наш API в качестве заголовка. И здесь, у вас может возникнуть соблазн сохранить его в локальном хранилище. Однако такой подход подвержен XSS-атакам.

Создание файлов cookie на клиенте для сохранения JWT также будет подвержено XSS. Если его можно прочитать на клиенте из Javascript вне вашего приложения — то его можно украсть. Вы можете подумать, что cookie с флагом HttpOnly поможет решить данную проблему, но файлы cookie уязвимы для атак CSRF.

Где же нам тогда его сохранить? И правильный ответ - в памяти фронтенд приложения.Итак, мы сохраним accessToken в памяти фронтенд приложении и, для этого, мы создадим специальную функцию, которую мы будем использовать для управления данным токеном. Давайте создадим папку services и в ней, мы добавим файл inMemoryJWT.js. Мы укажем здесь const inMemoryJWT = и далее объявим стрелочную функцию. Внутри нее мы создадим переменную inMemoryJWT, в которую мы поместим значение null.

Чуть ниже мы объявим функции getToken, которая будет возвращать данную переменную, а чуть ниже - setToken, которая пока будет принимать 2 параметра: token, который мы запишем в переменную inMemoryJWT и tokenExpiration (он нам понадобится чуть позже). И в конце, мы вернем данные две функции внутри объекта.

Экспортируем по умолчанию данную функцию и сразу вызываем.

Сейчас, в AuthContext, давайте импортируем эту функцию, и после успешного выполнения запросов на регистрацию и вход в аккаунт, мы деструктурируем значения accessToken и accessTokenExpiration и передадим их в функцию inMemoryJWTManager.setToken через запятую. В функции handleSignIn, мы проделываем то же самое.

**Защищенный роут**

Итак, следующим шагом, давайте реализуем логику запроса на защищиенный эндпоинт. Пока что, во вкладке "Демо", при нажатии на кнопку Запроса еще ничего не происходит. Давайте в файле server.js сделаем следующее, мы добавим новую конечную точку app.get("/api/protected") и в результате такого запроса мы вернем статус 200 с таким текстом, например: "Добро пожаловать!" и, чтобы при каждом запросе он отличался, в конце, мы добавим timestamp.

Сейчас во фронтенд приложении, мы добавим еще один экземпляр для axios, чтобы выполнять запросы к защищенным ресурсам. Мы укажем const ResourceClient = axios.create, и в свойстве baseURL укажем тот же url, что мы используем для авторизации, заменив auth на resource, свойство withCredentials здесь мы уберем, так как работа с куки в данных запросах нам не понадобится.

И теперь, в функции handleFetchProtected, мы вызовем данный запрос, Api.get("/protected"). И далее, мы запишем полученные данные в состояние data, которое в последвствии будет выведено на экран. Мы укажем setData и передадим в него свойство response.data.

**Заголовок Authentication**

И сейчас, давайте, вместо того, чтобы напрямую указывать здесь заголовок Authorization, мы создадим специальный interceptor. Что же это такое?

В axios существуют специальные interceptorы, с помощью которых мы можем «перехватить» запросы или ответы от сервера до того, как они будут обработаны. В данном случае, до того как запрос улетит на сервер, мы можем добавить в него заголовок.

Давайте чуть выше укажем ResourceClient.interceptors.request.use Данный код добавляет перехватчик для запросов к Api. Этот перехватчик будет вызываться перед отправкой каждого запроса с экземпляром ResourceClient.

Внутри перехватчика нам необходимо определить две функции:

Здесь первая коллбэк функция принимает объект `config`, который представляет конфигурацию запроса перед его отправкой. Внутри функции мы сделаем следующее:

Мы получим значение токена доступа из inMemoryJWT и сохраним в константе `accessToken`. Если токен отсутствует, то мы возвратим исходную конфигурацию запроса, без изменений. В противном случае, в запрос мы добавляем заголовок "Authorization" с помощью строки "Bearer " + accessToken. И наконец - возвратим измененную конфигурацию запроса.

Вторая коллбэк функция принимает параметр error. Здесь, внутри функции, мы вернем промис с отклоненным значением (`Promise.reject`) и передадим в него данную ошибку.

Таким образом, данный код добавит заголовок "Authorization" к каждому запросу, в случае, если accessToken содержится в памяти приложения..

**Проверка защищенного роута**

Прекрасно! Сейчас при нажатии на данную кнопку, все отрабатывает корректно. Запрос отправляется, сервер отдает нам данный текст, и мы отображаем его на экране. Однако, на данный момент, абсолютно любой пользователь может выполнять запрос к такой конечной точке. Поэтому, нам необходимо добавить middleWare на сервере, чтобы отслеживать случаи, когда пользователь, выполняющий данный запрос действительно является авторизованным.

**Валидация токена на бэкенде**

В классе TokenService содержится метод authorizeUser, с помощью которого мы будем проверять токен на валидность. И в случае ее успешного прохождения - вызывать соответствующий контроллер. Данный метод принимает 3 параметра - запрос, ответ и итератор next. И здесь, в файле server.js, мы укажем ее в качестве middleware перед вызовом контроллера для api/protected.

Внутри данной функции нам необходимо сделать следующее:

Из заголовка запроса `Authorization` мы извлекаем значение accessToken'a и сохраняем его в переменной `authHeader`.

Так как это токен на изъявителя (Bearer токен), то при его отправке, значение данного заголовка нам необходимо разделить с помощью метода `split(" ")`. Затем с помощью оператора опциональной цепочки, мы извлекаем сам токен и сохраняем в переменной `token`.

Если токен не был получен, то мы возвращаем статус ответа 401. Укажем next(и в него передадим new Unauthrorized()) Затем, в случае, если данный токен приложен к запросу, нам необходимо проверить его подлинность. Мы сделаем это с помощью метода `verify` из модуля `jsonwebtoken`.

В качестве аргументов мы передаем токен, секретный ключ из переменной окружения `ACCESS\_TOKEN\_SECRET` и колбэк-функцию, которая будет вызвана после проверки токена.

В колбэк-функции проверяем наличие ошибки. Если ошибка есть, то возвращаем статус ответа 403 (Forbidden), а в json отдаем полученную ошибку. Если же токен есть в заголовке, и его проверка прошла успешно, то мы вызовем функцию `next()`, чтобы перейти к выполнению контроллера.

**Тестирование защищенного роута**

Теперь, если мы вновь зарегистрируемся на сайте, укажем к примеру имя пользователя user3, пароль точно такой же, к примеру, - 123, а затем выполним запрос на защищенный роут, то в результате мы получим текст ответа, который был указан на сервере. Прекрасно!

**Выход из аккаунта**

Следующее, что мы реализуем - это выход из аккаунта. Давайте посмотрим, как его формально реализовать.

1. На стороне клиента выполняем запрос на /auth/logout c refreshToken в куки.

2. На сервере - удаляем запись из таблицы RefreshSessions по полученному refreshToken’у.

3. После выполнения запроса, на клиенте удаляем локально сохраненный в памяти accessToken.

AccessToken умрет по истечению срока его жизни. Пытаться его каким-либо образом инвалидировать не нужно, так как это нарушит суть JWT.

Итак, здесь, в контроллере logout, мы будем получать refreshToken из cookie, и далее, в блоке try - вызывать метод logout, расположенный в классе AuthService. Данный токен мы гарантированно получим здесь, так как в валидации он уже указан.

Далее, из Refresh сессий, нам необходимо удалить запись с данным рефреш токеном. Мы укажем здесь await RefreshSessions.deleteRefreshSession и передадим refreshToken в качестве аргумента.

В методе deleteRefreshSession мы укажем refresh токен в качестве параметра, и далее вызовем await pool.query и передадим запрос ("DELETE FROM RefreshSessions WHERE refresh\_token=$1) и, в массиве, передаем данный refreshToken.

И наконец, удалим из куки и сам refresh\_token. мы сделаем это в контроллере, перед отправкой данных обратно клиенту, мы укажем res.clearCookie(“refreshToken”).

Теперь, на стороне клиента, в функции handleLogout, нам необходимо отправить запрос на данный эндпоинт - /auth/logout. Мы укажем Auth.post(/logout) и после успешного выполнения запроса - удалим accessToken из памяти фронтенд приложения, при помощи inMemoryJWT.deleteToken. А в случае ошибки - выведем ее на экран.

**Тестирование разлогинивания**

Давайте обновим страницу и вновь зарегистрируемся, укажем к примеру имя user4, пароль 123 и роль - такую же - Администратор. Сейчас, после нажатия на кнопку зарегистрироваться, мы также можем получать защищенные данные на вкладке “Демо” и теперь, если мы нажмем кнопку выйти, то в панели разработчика, увидим запрос logout. Наш access\_token был удален из памяти приложения, а refresh\_roken - из файлов куки. При нажатии на кнопку "запрос на защищенный ресурс", теперь мы получаем статус 403.

Итак, на данный момент мы можем создавать и выходить из аккаунта, а также выполнять запросы к защищенным ресурсам. И сейчас, вместо того, чтобы создавать новый аккаунт каждый раз, давайте реализуем вход в аккаунт.

**Вход в аккаунт**

По сути дела, на фронтенде нам необходимо сделать почти то же самое, что и при регистрации. Давайте скопируем код функции handleSignUp и поместим его внутрь функции handleSignIn, изменив url с sign-up на sign-in. На стороне бэкенда, в метод signIn, мы пока что, поместим код контроллера signUp. Единственное, уберем поле role, при входе, мы не получаем его от клиента. Итак, для входа пользователя, нам необходимо реализовать следующую логику:

1. Пользователь входит в аккаунт, передавая на сервер имя и пароль.

2. Генерируем на сервере fingerprint браузера

3. Проверяем, существует ли в пользователь с данным именем в базе данных.

4. Если такого пользователя не существует - выбрасываем ошибку.

5. Хэшируем введенный пароль

6. Сравниваем хэш пароля в базе данных с хэшем введенного пользователем пароля

7. Если хэши не совпадают - выбрасываем ошибку

8. Генерируем access и refresh токены

9. Добавляем рефреш-сессию в базу данных (поля: id пользователя, refresh токен, fingerprint браузера) (об этом немного позже)

10. Записываем refresh токен в httpOnly cookie

11. Возвращаем клиенту access токен

Итак, здесь, в методе signIn класса AuthService, мы деструктурируем поля userName, password и fingerPrint.

И здесь, в данном методе, мы первым делом добавим проверку на то, что данный пользователь существует. В случае, если пользователь с данным именем отсутствует, мы выбросим исключение NotFound, с текстом ошибки, к примеру: “Пользователь не найден!”.

Следующее, что нам необходимо сделать - это сравнить хэши паролей. Давайте создадим константу isPasswordValid, равную bycrypt.compareSync, чтобы сравнить хэш введенного пользователя пароля (hashedPassword) с хэшем пароля, хранящегося в базе данных (userData.password).

Если данные два хэша отличаются, следовательно пароль неверный. И в данном случае нам так же необходимо выбросить исключение BadRequest, с текстом, например “Неверный логин или пароль”.

Далее, в свойства role и id объекта payload записываем значения из объекта userData. Код, расположенный ниже, оставляем в таком виде. Мы генерируем пару токенов с данным payload и создаем рефреш сессию в базе данных, а в конце - возвращаем данную пару токенов и время жизни для access токена.

**Тестирование входа в аккаунт**

И теперь, если мы нажмем на кнопку выйти из аккаунта и затем перейдем на страницу для входа в аккаунт, введем ранее созданное нами имя пользователя user4 и пароль 123, а затем нажмем на кнопку “Войти”, то мы успешно войдем под данной учетной записью, а в таблице refreshSessions появится запись о данном пользователе.

Сейчас, мы также имеем и доступ и к защищенному ресурсу. Пока что все работает прекрасно. Мы можем регистрироваться, а также входить и выходить из аккаунта. Однако сейчас у нас есть небольшая проблема. Первые 30 минут после входа в аккаунт, мы беспрепятственно можем выполнять запросы на защищенные эндпоинты и получать от них ответ. Но спустя 30 минут после входа, то есть после того, как истечет время жизни access токена, мы больше не сможем получать такие данные. Для решения этой проблемы, мы реализуем механизм обновления access токена.

**Рефреш**

Конечно, есть несколько способов обновления токена на клиенте. Это можно сделать, например, при помощи интерсептора, который при получении статуса 403 отправит запрос на рефреш токена. Однако в данном проекте, мы реализуем такой механизм немного по-другому.

В файле inMemoryJWTService, мы добавим функцию refreshToken, которая в качестве параметра будет принимать длительность жизни токена в миллисекундах. Внутри данной функции, мы будем запускать таймер обратного отсчета, по истечению которого, мы будем обращаться к серверу за новым токеном.

Мы добавим константу timeoutTrigger, в которую мы запишем значение expiration - 10000, чтобы наш таймер срабатывал немного раньше истечения жизни токена, в данном случае - на 10 секунд.

Затем, чуть выше мы объявим идентификатор для нашего таймера, который мы назовем refreshTimeoutId, и, в самой функции, в данное значение refreshTimeoutId, мы поместим setTimeout. Внутри мы укажем callback функцию, которая будет вызвана по прошествии данного количества миллисекунд, равного timoutTrigger.

Итак, по истечении timoutTrigger, внутри данного коллбэка, мы вызовем AuthClient.post(“/refresh”). И в случае успешного выполнения запроса, мы вызовем тот же код, что и при входе в аккаунт и регистрации - мы деструктурируем значения accessToken и accessTokenExpiration, полученные от сервера, которые, мы впоследствие, передадим функции setToken.

Все что нам теперь остается сделать на клиентской части - это добавить вызов фукции refreshToken здесь, внутри setToken, чтобы запустить данный таймер при получении токена, а также, в файле AuthContext, вызвать refresh при первом запуске приложения. Мы добавим useEffect, внутри которого, сразу после монтирования компонента, мы обратимся к серверу за токеном.

Прекрасно! Давайте перейдем к логике обновления токена на бэкенде. Процесс рефреша на сервере можно описать формально следующим образом:

1. Получаем на сервере запись рефреш-сессии по рефреш токену.
2. Сохраняем текущую рефреш-сессию в переменную и удаляем ее из таблицы
3. Выполняем 2 проверки:

i. На истечение срока жизни refresh токена

ii. На соответствие fingerprint, полученного из текущей рефреш-сессии с fingerprint, полученным в запросе.

1. В случае негативного результата мы отклоним такой запрос и вернем клиенту статус 403 - forbidden.
2. В случае успеха:

i. Генерируем новые access и refresh токены

ii. Создаем рефреш-сессию на основании fingerprint и созданного refresh токена и записываем ее в БД

1. Отправляем клиенту access и refresh токены

Возможно сейчас такая последовательность действий кажется вам немного непонятной. Давайте перейдем к ее реализации, в процессе которой, мы проясним некоторые моменты.

Важной особенностью в таком алгоритме является то, что в момент рефреша, обновляются ОБА токена. И тут возникает вопрос. Как refresh токен может сам себя обновить, если он создается только после успешной аутентификации? refresh token в момент рефреша сравнивает себя с тем refresh token'ом который находится в БД и в случае успеха, а также если у него не истек срок, мы рефрешим оба токена.

Вопрос зачем refresh token'y срок жизни, если он обновляется каждый раз при обновлении access token'a? Это сделано на случай, если пользователь будет в офлайне более 15 дней. В этом случае, такому пользователю придется заново указать логин/пароль для входа.

Внутри контроллера, мы получаем refreshToken из req.cookies и fingerPrint из тела запроса req.body. Затем, мы вызовем метод AuthService.refresh() и передадим оба данных значения в объекте.

Затем, на основе refresh токена, мы извлечем рефреш-сессию из хранилища RefreshSessionsRepository. Если данной рефреш-сессии не существует (!refreshSession) или, ее фингерпринт отличается от того, что мы получаем в данном запросе (то есть refreshSession.finger\_print !== fingerPrint), мы выбросим исключение Forbidden.

Затем, мы удалим данную рефреш-сессию из таблицы. Мы вызовем await RefreshRepositories.deleteRefreshSession и передадим в данный метод полученный refreshTokken.

Далее, используя функцию jwt.verify и REFRESH\_TOKEN\_SECRET из переменной окружения, мы проверим данный токен на валидность. Данный участок кода мы обернем в try/catch и в случае невалидности токена - также вернем статус 403. Данный метод вернет нам расшифрованный payload, который мы будем использовать для генерирования новых токенов.

Генерируем новую пару токенов с помощью методов generateAccessToken и generateRefreshToken.

И наконец, создаем новую рефреш-сессию. В качестве id пользователя мы передаем payload.id, в refreshToken - значение newRefreshToken, а также fingerPrint данного пользователя.

В качестве результата работы данного метода, мы вернем объект, содержащий новые access и refresh токены, а также значение accessTokenExpiration.

Как итог в контроллере мы запишем значение refreshToken в куки, а в json - вернем значения accessToken и accessTokenExpiration.

**Привести фронт в порядок**

**Защита от краденного токена**

**Авторизация с нескольких устройств**

Теперь, благодаря поддержке рефреш-сессий, мы можем авторизоваться с нескольких браузеров и для каждого такого браузера, хранится отдельная рефреш-сессия в базе данных.

Однако, стоит заметить, что процесс добавления сессии в таблицу должен имеет свои меры безопасности. При добавлении стоит проверять сколько рефреш-сессий всего есть у пользователя и, если их слишком много или пользователь входит одновременно из нескольких подсетей, стоит предпринять меры.

К примеру, можно добавить проверку, что пользователь имеет максимум до 5 одновременных рефреш-сессий, и при попытке установить следующую, удаляем предыдущие. Однако данная проверка также по усмотрению, в зависимости от решаемой вами задачи.

На этом у меня все!